programming with R

adrian alexa
aalexa@illumina.com

this talk is about

a list of useful R tricks (hopefully)

a collection of problems I had to solve

code optimization and profiling

not in this talk

~~an introduction to R~~ (not for complete beginners)

~~a set of rules one needs to follow~~

~~object oriented programming~~

~~graphics, statistics, modeling, or other cool stuff one usually does with R!~~

programming with R

vs.

working with R

Patrick Burns made a point that the writing R functions requires a different style than working interactively with R.

```
myMat[, 1]
myMat[, 1, drop = FALSE]
```

i never use `apply()` **or** `sapply()` **in my code.**
plenty of `lapply` **and** `for/while/repeat` **loops.**

problems of R ?

R is an interpreted language – thus show !!!

R doesn't directly operate with the underlying data types. it works on abstract representations of those (think at them as structures in C).

there is a wide spread believe that R code is very slow, around 100 times slower than native C code.

quick tips

try to use as many primitive functions as possible.

use matrices or lists whenever is possible. matrix computation in R can be faster even than naiive C code.

avoid doing computations directly on large data.frames. they are a lot slower compared to matrix computations.

pre-allocate the space for the entire array/matrix when creating the object and use column assignments whenever possible.

quick tips

```
m <- do.call(cbind, l)
```

vs.

```
for(i in 1:length(l))
    m <- cbind(m, l[[i]])
```

```
identical(x, y)
```

for checking the equality
of two R objects

```
x[which(x > 0)]
```

vs.

```
x[x > 0]
```

sometimes using `which()` will be faster! **why**? but be aware of `x[-which(expr)]`

```
unlist(l, use.names = FALSE)
```

one seldom use the list names
and the speedup is tremendous

code profiling

when the code works correctly, the next step is to find out which parts are too slow, and try to speed them up.

this requires measurement, rather than guessing and R provides the basic tools for performance analysis.

the R profiler won't tell you the complete story!
(native function can allocate system memory which is not traceable by R)

code profiling

`system.time()`	system time measurements
`Rprof()`	starts/stop the R profiler
`summaryRprof()`	display profiling data
`Rprofmem()`	function for profiling R memory usage (needs build in support)

there are two extra packages: profr and proftools. they offer functions for plotting the results

code profiling

```
> Rprof("myRprofile.out", memory.profiling = TRUE)
your code here ....
> Rprof(NULL)

> summaryRprof(myRprofile.out)
Time and memory stats of your code ....
```

let's use the profiler to check what is the difference between `for` loops and functional iterators like `lapply()`, `apply()`, `by()`...

code profiling

```
> nr <- nc <- 1e4L

> Rprof(interval = 0.001, memory.profiling = TRUE)

> set.seed(1234)
> mat <- matrix(0L, nrow = nr, ncol = nc)
> for(i in seq_len(nr))
    mat[i, ] <- sample(nc)

> Rprof(NULL)
> summaryRprof(memory = "both")
```

code profiling

```
$by.self
         self.time self.pct total.time total.pct mem.total
"sample"     0.513    76.34      0.513     76.34      40.8
"matrix"     0.159    23.66      0.159     23.66      47.7

$by.total
         total.time total.pct mem.total self.time self.pct
"sample"      0.513     76.34      40.8     0.513    76.34
"matrix"      0.159     23.66      47.7     0.159    23.66

$sample.interval
[1] 0.001

$sampling.time
[1] 0.672
```

code profiling

```
> mean(replicate(10, system.time({set.seed(1234)
    mat <- matrix(0L, nrow = nr, ncol = nc)
    for(i in seq_len(nr))
      mat[i, ] <- sample(nc)
  })["elapsed"]))
[1] 7.8444
```

`system.time()` gives the real time for the evaluation of the expression.

```
> mean(replicate(10, system.time({set.seed(1234)
    mat <- matrix(0L, nrow = nr, ncol = nc)
    for(i in seq_len(nc))
      mat[, i] <- sample(nr)
  })["elapsed"]))
[1] 3.4799
```

code profiling

```
> mean(replicate(10, system.time({set.seed(1234)
    matA <- matrix(nr, nrow = 1L, ncol = nc)
    matA <- apply(matA, 2, sample)
  })["elapsed"]))
[1] 2.7831
```

`apply()` version of the problem

```
$by.self
          self.time self.pct total.time total.pct mem.total
"FUN"         0.452    63.75      0.452     63.75     157.4
"array"       0.147    20.73      0.147     20.73      47.7
"unlist"      0.084    11.85      0.087     12.27      50.1
"apply"       0.024     3.39      0.709    100.00     262.3
"lapply"      0.002     0.28      0.003      0.42       2.3
```

code profiling

```
> mean(replicate(10, system.time({set.seed(1234)
   mat <- do.call(cbind, lapply(rep.int(nr, nc),
sample)) })["elapsed"]))
[1] 2.2558
```

unlisting

assume we need to compute a statistic over all elements of a list

for example, to count how many distinct elements are in the list

> "*given a list structure* `x`, `unlist()` *simplifies it to produce a vector which contains all the atomic components which occur in* `x`."

unlisting

```
> length(id2GO)
[1] 18490

> str(id2GO[1:5])
List of 5
 $ ENSG00000124209: chr [1:18] "GO:0007032" "GO:0006897"…
 $ ENSG00000064703: chr [1:22] "GO:0000244" "GO:0008026"…
 $ ENSG00000171408: chr [1:6] "GO:0016787" "GO:0004114"…
 $ ENSG00000187223: chr "GO:0031424"
 $ ENSG00000213523: chr [1:21] "GO:0005737" "GO:0006915"…

> str(unlist(id2GO))
 Named chr [1:156036] "GO:0007032" "GO:0006897"…
 - attr(*,"names")= chr [1:156036] "ENSG000001242091" …

> length(unique(unlist(id2GO)))
[1] 7413
```

unlisting

```
> Rprof(interval = 0.01, memory.profiling = TRUE)
> for(i in 1:1000)
     invisible(unlist(id2GO))
> Rprof(NULL)
> summaryRprof(memory = "both")$by.total
         total.time total.pct mem.total self.time self.pct
"unlist"     114.39       100     147.3    114.39      100
```

`unlist()` tries to keep the naming information present in `x`, and the resulted names are mangled by default –> poor performance.

```
> Rprof(interval = 0.01, memory.profiling = TRUE)
> for(i in 1:1000)
     invisible(unlist(id2GO, use.names = FALSE))
> Rprof(NULL)
> summaryRprof(memory = "both")$by.total
         total.time total.pct mem.total self.time self.pct
"unlist"       5.91       100      73.3       5.91      100
```

inverting mappings

an interesting problem is to
inverse a `list` of mappings

given a mapping form genes to GO terms as a list, compute which are the genes mapped to each GO

inverting mappings

```
inverseList <- function(l) {
   ##num.rId <- sapply(l, length)
   num.rId <- unlist(lapply(l, length), use.names = FALSE)

   rId <- unlist(l, use.names = FALSE)
   lId <- rep.int(names(l), num.rId)

   return(split(lId, rId))
}

> str(inverseList(id2GO)[1:3])
List of 3
 $ GO:0000002: chr [1:3] "ENSG00000025708" "ENSG00000151729" …
 $ GO:0000003: chr "ENSG00000189409"
 $ GO:0000009: chr "ENSG00000182858"
```

indexing

indexing is very fast in R, but it is quite a generic operation and not all cases are optimized!

```
x[10:1000]
```

what happens when the following code gets evaluated?

indexing using IRanges

`window()` method from **IRanges** (bioconductor pakage) provides some cool functionality.

```
window(x, start = 10L, end = 1000L)
```

why is this better?

there is some penalty for being a method though!

indexing using IRanges

```
> x <- seq_len(1e8)
> st <- 10L
> en <- length(x) - 10L

> system.time(x1 <- x[st:en])
   user  system elapsed
  0.964   0.400   1.363

> system.time(x2 <- window(x, start = st, end = en))
   user  system elapsed
  0.232   0.128   0.360

> identical(x1, x2)
[1] TRUE
```

some other tips

one can use `as.vector(IRanges())` to generate vector indices - quite fast for matrix indexing `mat[st:en, ]`.

use `tabulate()` instead of `table()` when you just want the counts. `Rle()` can also be used if the data is ordered!

a combination of `scan()` and `readBin()`/`readChar()` can give you very fast IO, much faster than `read.table()`.

high performance computing

mmap package is really nice if you work with large external arrays – lets you build a custom columnar database

rmongodb lets you leverage the power of mongoDB system

scidb leverages the power of SciDB array-oriented database. keep an eye on this!

things to take home

speedup is possible in R using a few tricks. sometimes are easy to implement, sometimes they are not.

start with simple approaches familiar to you. once the code is functional you can start the optimization.

the choice of data structure is critical to the performance. use vectorization whenever its possible.

things to take home

do not grow objects and avoid recopying large objects.

transform the data to a user friendly object at the end of the function, see `data.frames`

`for` loops are not slow, but the operations performed inside are. have a compromise between `for` loops and `*apply()`.